2014 IEEE International Conference on Internet of Things (iThings 2014), Green Computing and Communications (GreenCom 2014), and Cyber-Physical-Social Computing (CPSCom 2014)

Global-scale Federated Access to Smart Objects Using XMPP

Daniel Schuster, Philipp Grubitzsch
Computer Networks Group
TU Dresden, Germany
(daniel.schuster, philipp.grubitzsch)@tu-dresden.de

Dominik Renzel, István Koren
ACIS Group
RWTH Aachen University, Germany
(renzel, koren)@dbis.rwth-aachen.de

Ronny Klauck, Michael Kirsche
Computer Networks Group
BTU Cottbus-Senftenberg, Germany
(klaucron, kirscmic)@tu-cottbus.de

***Abstract*—Communication with smart objects currently only works in isolated, sometimes even proprietary islands. This lack of interoperability limits the value of smart objects connected to the Internet of Things (IoT). We propose to use the eXtensible Messaging and Presence Protocol (XMPP) to connect IoT islands. XMPP is inherently federated, secure, globally scalable and designed for collaboration. We use XMPP Multi-User Chat (MUC) to build a secure and accessible platform for sensor data exchange between organizations. We demonstrate a scenario of three distributed and interconnected XMPP-driven sites, whereas different client types access sensor data from all sites. Our evaluation results confirm that the architectural pattern presented in this work can easily be used in any XMPP-based system without the need to enhance or to extend the standards.**

***Keywords*—XMPP; Internet of Things; Federation.**

I. Introduction

The value of the *Internet of Things (IoT)* will only grow with the number of devices able to communicate with each other. According to Metcalfe's law, the value of a network is proportional to the square of the number of connected devices.

Unfortunately, a real *Inter*net of Things does not exist yet. Many smart objects still ship with proprietary smartphone or tablet apps. This strategy creates thousands of isolated smart object islands. Furthermore, this "one-app-per-object" approach [1] does not scale with the number of devices *one* single user interacts with.

Integrating diverse device types into a comprehensive network of smart objects thus remains a grand challenge. With particular regard to interoperability, a true IoT must allow users to access smart objects from any vendor with their mobile devices from a possible different vendor without considering how interconnection and communication works.

The World Wide Web often comes into mind as an analogy. Accessing websites with the HTTP protocol is simple and efficient, but it only works if the device hosting the website is registered in the DNS. Such registration requires administrative effort. Thus, an interoperability solution for smart objects has to support easy and decentralized object discovery and secure information access and control for objects by end-users. Last but not least, end users and developers should be enabled to interact with this IoT using arbitrary, well-established front-end technologies like Web applications or simple apps.

We employ the *eXtensible Messaging and Presence Protocol (XMPP)* to realize the listed requirements. XMPP was explicitly designed as a federated platform for different types of authenticated entities (end-users, automated agents, etc.). Each entity can own an arbitrary number of resources (e.g., devices, smart objects). XMPP has native support for entity discovery as well as real-time communication and presence between entities and resources. It is the only open standards family available today that offers inherent and extensible support for global-scale communication among different entities.

This work proposes a simple, light-weight approach. An *XMPP Multi-User Chatroom (MUC)* serves as an efficient metaphor to model a smart object or a place in the real world, where sensors are present and push information or receive commands. XMPP MUCs are ready-to-use, as they support discovery and fine-grained access control. Federated access is simple and can be restricted to certain domains.

The remainder is structured as follows. In Section II, we state the problem definition. Existing solutions are summarized in Sections III and IV, respectively. Our solution approach to use XMPP MUCs is introduced in Section V. In Section VI, we demonstrate the feasibility of our approach with the ACDSense scenario, where three formerly independent sites are integrated into a federated sensor network. First evaluation results for the example scenario and our approach are presented in Section VII. We conclude our work in Section VIII.

II. Problem Definition

The problem definition seems to be quite simple: Every device in the IoT should be able to discover and securely access every other device to retrieve information and send commands. The device owner should be in full control of who is accessing his devices. However, to the best of our knowledge, there is currently no work solving this problem completely.

First, we have to differentiate types of devices and their communication patterns as shown in Figure 1.

Three main communication patterns need to be supported:

Level 1: *Any UI device is able to access information collected by cloud services from smart objects*
The first level seems to be easily solvable by adding REST-like APIs to current cloud services. Fine-grained access control is possible, but has to be repeatedly set up for every provider of cloud-controlled smart objects. The information flow is $n:1:m$. An arbitrary number of devices is able to access the information using an intermediate node (the cloud service). Federation can be added as an extension (Level 1+), if cloud services are able to talk to each other using similar APIs.

978-1-4799-5967-9/14 $31.00 © 2014 IEEE
DOI 10.1109/iThings.2014.35

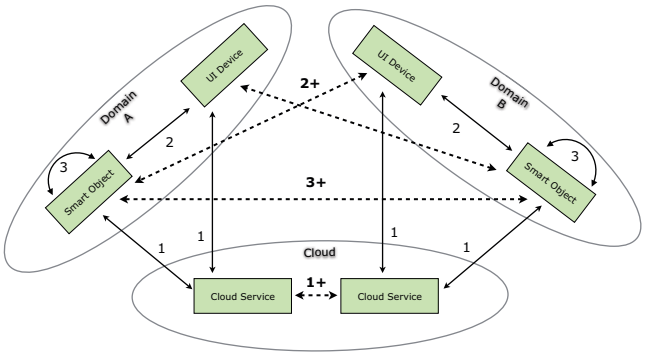

Fig. 1. Types of devices and communication patterns

Level 2: *Any UI device is able to access smart objects directly* The second level calls for easy discovery mechanisms, like showing all smart objects in a room or house. Access control is harder to achieve here, but still bound to a user. Communication flow is $1 : n$. Multiple UI devices may access a smart object simultaneously. Inter-domain communication is depicted as Level 2+ in Figure 1. This is harder to achieve than Level 1+, because device discovery, actual communication and access control must work in a federated fashion.

Level 3: *Smart objects are able to identify and communicate with each other to coordinate*
Level 3 discovery is similar to Level 2 discovery. However, access control is more complex, because this level loses the user metaphor. Smart objects can be both attached to a user as well as simply attached to a room or larger movable objects. Information flow can upgrade to $n : m$; i.e., each object can possibly send individual messages to each other object belonging to the same group (place). Again, the inter-domain variant is called Level 3+.

Regarding the communication patterns, we typically expect a manageable amount of small messages to be exchanged at the last mile, while the flow of sensor data can grow to large streams between communication domains. We argue that XMPP is suitable to support all of the above patterns, which we will elaborate in the following sections.

III. XMPP AND THE IOT

A. Introducing XMPP

Originally designed as an Instant Messaging protocol, XMPP inherits the extensible properties of XML, the message format used to transport diverse types of small messages *(stanzas)* between arbitrary entities in near real-time. Messages are sent via an XMPP server to so-called *JIDs*. A *Bare JID* usually represents a user (e.g., *user@mydomain.org*), while a *Full JID* represents a resource owned by the user (e.g., *user@mydomain.org/phone*).

Messages may be sent either to a concrete resource (using the Full JID) or to a user (via the Bare JID). In the latter case, the actual receiving resource is determined by priorities which can be set by sending presence notifications.

Like e-mail, XMPP is designed as a federated system. Usually, there is one XMPP server per organization as well as some independent servers which offer free or paid use of their services for private persons. These servers interconnect via DNS to relay messages to the target domain.

More information on XMPP can be found in [2] and at the website *xmpp.org*. An active community constantly extends the capabilities of XMPP and provides free server, library, and client implementations for a large number of platforms. Official extensions are specified and managed by the *XMPP Standards Foundation (XSF)*. These extensions enable additional functionalities like multi-user chat, file sharing, and serverless messaging. The full list of official extensions is available at *xmpp.org/extensions*.

B. IoT-related Activities

Using XMPP for IoT scenarios is part of ongoing specification activities. Current approaches can be split in two opposed strategies: **(i)** integrate smart objects with XMPP-enabled gateways and **(ii)** directly deploy XMPP on resource-constrained devices.

For **(i)**, works like [3]–[5] use the XMPP Publish-Subscribe extension in a mediated way: non-constrained devices (e.g., desktop, smartphone, netbook) act as gateways, translating sensor-specific data into XML messages, while the smart object itself is not directly connected to the XMPP network.

[3] covers context management for sensors and introduces an XML privacy scheme to protect sensor data in XMPP networks. Gateways are here used to connect sensor devices. Sensor Andrew [5] is a large-scale framework to deploy sensors across Carnegie Mellon University and to prototype scalable applications. [4] integrates sensor data from Android smartphones directly in an XMPP network by deploying a smartphone-based XMPP client and by coupling XMPP and the Java-based OSGi framework to realize context-aware data processing in industrial M2M scenarios.

In addition, dedicated XMPP extensions for IoT scenarios are currently being developed by Peter Waher [6]. They focus on large-scale M2M communication for industrial environments. Sensors are accessed using a *Concentrator* node that acts as an XMPP client. Fine-grained access control can be reached by involving a so-called *Provisioning Server* as part of the XMPP network.

Approaches of **(i)** break the Internet's end-to-end principle by relying on gateways / protocol translators. In contrast, approaches from **(ii)** focus on a direct deployment of XMPP on constrained devices and the adaption of XMPP for use in scenarios with resource-constrained devices.

A first step in this direction is μXMPP [7], which demonstrates that XMPP can run on constrained devices. μXMPP includes a reduced set of XMPP core functionalities and enables a direct communication between smart objects and non-constrained devices. Chatty Things [8] improves μXMPP with: an API for developing IoT applications, support for essential extensions, optimizations for small RAM/ROM footprints, and a reduction of XMPP message exchange. The custom XMPP extension *Temporary Subscription for Presence (TSP)* [8] reduces the number and size of messages for joining a MUC room. The constrained object joining the MUC room only

announces its presence and does not receive MUC presence in return.

Chatty Things first introduced the idea to send sensor data to an XMPP MUC room. This inherently simple concept is powerful enough to be extended for federated and discoverable smart object communication, as shown in the next section.

C. Discussion

XMPP has already been recognized as a viable solution by some works from the IoT domain. Communication can be either direct, via the Publish-Subscribe extension or via MUC. Refer to Table I for a comparison of these approaches.

TABLE I. COMPARISON OF XMPP-BASED APPROACHES

	Direct	PubSub	MUC
1:1 communication	X	X	X
1:n communication	–	X	X
Topics (n:1)	O	X	X
Tree of topics	–	O	–
Discovery	O	X	X
Presence	O	–	X
Access control	O	O	X

– no support O limited support X good support

While all approaches handle the basic case of 1:1 communication quite well, only PubSub and MUC are able to distribute data from smart objects to multiple receivers simultaneously. In the IoT, there is often a need to bundle multiple smart objects as a group or topic, e.g., all sensors belonging to a house. This can be done with PubSub and MUC, as multiple entities may publish on a PubSub node and MUC room, respectively. Grouping can be achieved via Concentrator nodes for the direct approach, but this only creates a static mapping of smart objects to a JID.

If the number of topics grows, there will be a need to cluster topics in a hierarchy. While this should theoretically be a pro argument for the PubSub approach, current implementations often omit the hierarchy aspect of the Publish-Subscribe extension. Till this day, we did not find a working combination of XMPP client library and XMPP server supporting hierarchical PubSub.

Before the actual information can be accessed, smart objects need to be discovered. For the direct approach this is only possible if the smart objects are using the same XMPP account as the requesting client. This may be the case for some objects in the smart home environment, which are tied to one common user account. PubSub and MUC both use the Service Discovery extension of XMPP which enables to retrieve a list of topics (`disco#items`) as well as to request details of every item (`disco#info`).

Online presence information is helpful to check the availability of smart objects. With the direct approach this is again only viable for smart objects using the same user account. Otherwise, the user needs to do a presence handshake with every object he is interested in. PubSub fails completely at this issue, as publishers and subscribers are decoupled. MUC offers the best presence support, as smart objects joining a MUC publish directed presence to this MUC, which can be consumed by every participant joining the MUC. Thus, smart object presence can be observed as easy as joining a MUC and leaving it after the observation has finished.

A last, important point is securing access to information from smart objects. For the direct case, the smart objects can use black- and white-listing on the server to filter communication. But these requests can only be issued by the XMPP user (i.e., the smart object) itself, making it hard to manage multiple objects. PubSub should offer fine-grained access control for topics, but again this aspect of the protocol is not implemented by most client libraries and server implementations (as of April 2014). The MUC approach offers the best support for access control. A role concept enables the management of black and white lists for multiple topics (i.e., MUC rooms) at once.

These arguments accounted for our decision to leverage the MUC-based approach for our IoT scenarios. We also compared it to other non-XMPP approaches, as described in the following section.

IV. NON-XMPP APPROACHES

XMPP is only one of the main competitors to act as the discovery and access protocol for the IoT. Other approaches are REST-like communication (HTTP/CoAP) and MQTT.

The intuitive approach for accessing sensors is to use REST-like HTTP GET requests and return the value of the sensor in JSON or pure text format. While this is one of the most prominent approaches, the smart objects requires a mini Web server or a proxy performing this operation on their behalf. A new TCP connection needs to be established and closed for each request. Compared to this, XMPP only requires the node to run a client library opening an outgoing TCP connection to its XMPP server. This connection is reused for many requests and responses. Even Web applications can follow this paradigm with XMPP over WebSockets.

The *Constrained Application Protocol (CoAP)* [9], [10] defines a UDP-based access protocol for smart objects, in particular for constrained devices with approximately 10 KB of RAM and roughly 100 KB of code space (class-1 devices). It uses HTTP-like pull semantics to access information on smart objects, but request and responses are transported using a compact encoding on top of UDP. Thus, smart objects do neither need to manage TCP connections nor an HTTP stack. Push semantics can be enabled by setting the Observer option, where the smart object will continue sending results to the requester as soon as the requested value changes. Discovery of sensors has to be done in a Web-like way by requesting a well-known URI (`GET /.well-known/core`), offering a collection of links.

While both HTTP and CoAP offer cross-domain access based on the DNS, access control has to be handled by other means. This approach is thus only practical to realize Level 1+ communication (compare Section II). For Level 2+ or Level 3+, access control is hard to achieve, as it has to be done for each UI device or even smart object individually. XMPP offers the unique advantage that all users are authenticated at their XMPP domain, and that access control can be done per domain or with additional black/white lists for individual users (authenticating all devices owned by the user). Furthermore, discovery of resources is more flexible with our MUC-based approach. The smart object itself is able to modify the registered information by managing the MUC room. Regarding the focus on constrained devices, Chatty Things has proven that

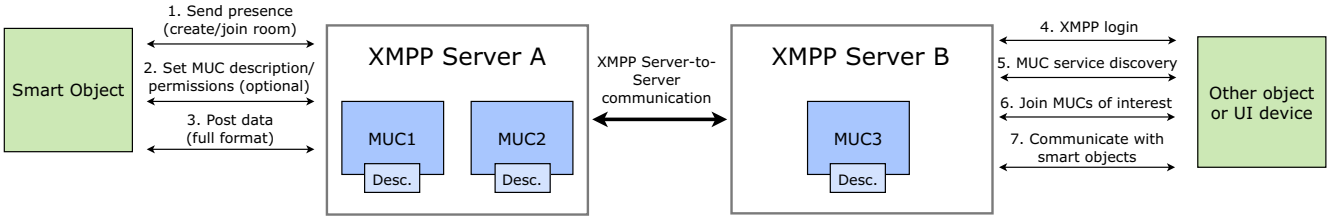

Fig. 2. Concept based on XMPP Multi-User Chat (MUC)

a "lightweight" XMPP is even able to run on class-1 devices. Finally, one advantage of our approach compared to CoAP is the simple support of multiple users accessing the same sensor information. The sensor only needs to push this information to the MUC room once, regardless of the number of receivers.

MQTT [11] uses a publish-subscribe architecture. Clients connect via TCP to a broker and subscribe to topics. A hierarchical topic scheme is used. Sensors may be modeled as topics and thus distribute their data. MQTT uses an efficient binary message format and offers different message delivery primitives (at-most-once, at-least-once, exactly-once). A modified version designed for wireless sensor networks is available as MQTT-SN [12]. It uses UDP instead of TCP and shorter topic identifiers to optimize MQTT for constrained devices.

While MQTT is the most efficient approach especially in one-to-many scenarios, it lacks federation capabilities. MQTT and other PubSub architectures are most useful in isolated application scenarios, where one central message broker is sufficient and federation not required.

V. The MUC-based Approach

Our approach of MUC-based sensor discovery, control and data retrieval is depicted in Figure 2. To the extent of our knowledge, it is the best fit to realize federated communication between smart objects and UI devices, i.e., the communication levels 2 and 2+ from Figure 1. To realize this approach, the XMPP server and client libraries need to support XMPP Multi-User Chat (XEP-0045) and Service Discovery (XEP-0030). This is the case for most servers and libraries in productive use. Each MUC room defines a description field, qualifying it as sensor MUC room.

A. Discovery and Communication

A smart object announces itself by sending presence to a MUC room. If this room does not exist yet, it is created by the MUC extension automatically. Multiple smart objects may join the same room. It is thus possible to create a room that represents a place or another real-world object with multiple sensors or smart objects.

The owner (usually the smart object that created it) can change description and permissions of the MUC room. The description is later used by other objects to discover the sensor MUC room. Permissions are used to restrict access to the MUC room on a coarse-grained (password protection) or a fine-grained basis (black-/white-listing).

Finally, the smart object sends its sensor data to the MUC room within the body of a message stanza. We propose to use *JavaScript Object Notation (JSON)* as the data format. It enables easy parsing, simple representation of data structures, and integration with Web technologies. Mixing up XML and JSON seems to be strange in the first place, but it is quite reasonable. The XML markup works as the data envelope, while the more native data format contains the actual payload.

To discover sensor MUC rooms at an XMPP server, an XMPP entity is first required to log into its XMPP server. It is then able to retrieve a list of MUC rooms at the XMPP server using the `disco#items` request of XEP-0030. It has to check whether the description of a MUC room marks it as a sensor MUC room and finally join the rooms of interest. The entity will then not only receive actual sensor data, but also a configurable window of historical sensor data.

The discovery described above not only works for the XMPP server the user is connected to, but also in a federated manner on remote XMPP servers. The same holds true for joining MUC rooms. Thus, discovery and actual access of sensor data across organizational borders is possible.

B. Sensor Metadata & Data

Given its ease-of-use and widespread adoption in Web applications, we employ all descriptions of sensor metadata and data in JSON. We propose the following JSON sensor metadata format, specified as a sensor MUC room description:

```
{"sensormuc":
 {"type":"MULTI",
  "format":"full",
  "location":
   {"countryCode":"US",
    "cityName":"Pioneer Village",
    "latitude":40.98520,
    "longitude":-111.9020}}}
```

If a MUC room carries a JSON string in its description field with key = `"sensormuc"`, it should be identified as a MUC room carrying sensor data. The type `"MULTI"` in the example states a MUC room that represents multiple sensors. In this case, different sensors pushing values to the MUC room have to provide their type in each sensor event.

Sensor data is specified as a JSON string in the context of a sensor event. The format is based on the Android API for sensor events. For the sake of simplicity, we prefer such a format over more complex formats like SensorML.

```
{"sensorevent":
 {"type":"AMBIENT_TEMPERATURE",
  "values":[20.34432],
  "timestamp":"2013-09-18T18:31:38+01:00"}}
```

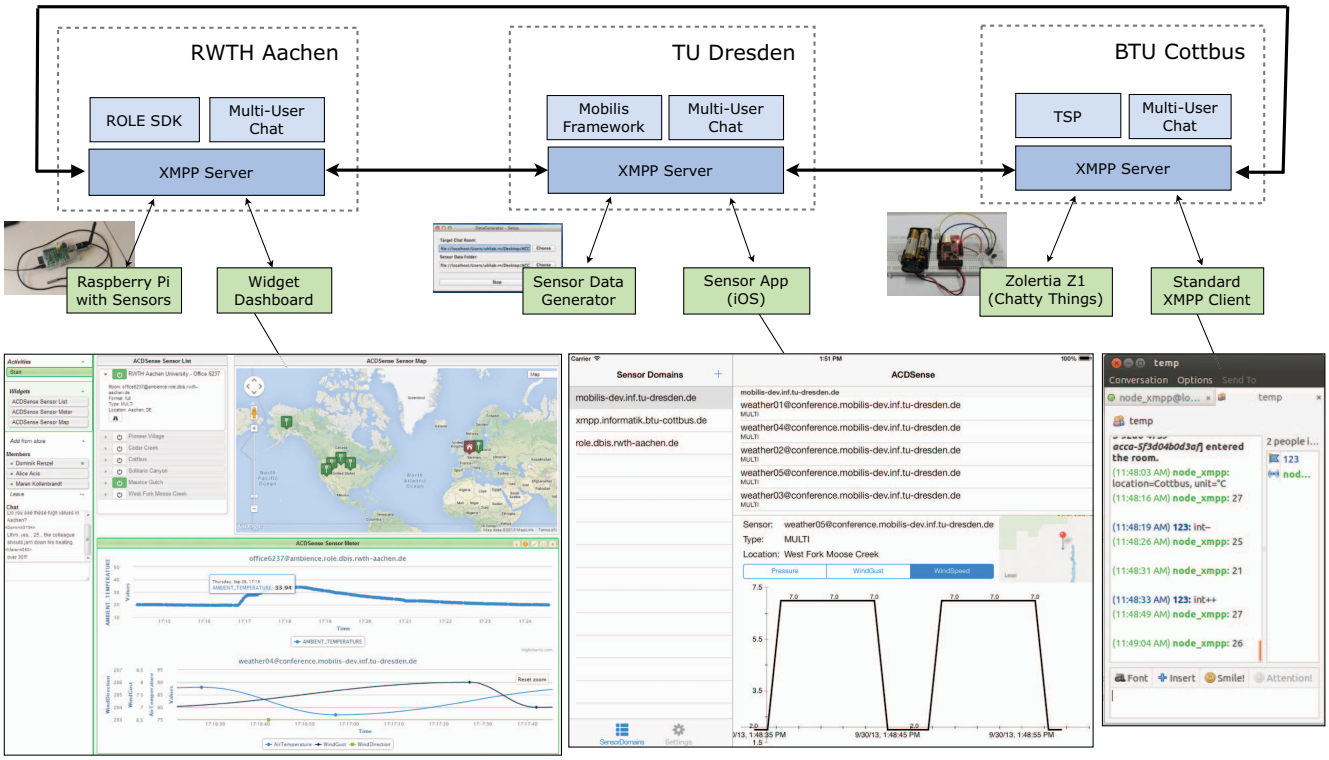

Fig. 3. Architecture and screenshots of the example scenario ACDSense

In cases of one sensor per MUC room, we collect all sensor metadata in the MUC room description and use a short format for the actual sensor data. This short format is especially useful in scenarios with resource constrained smart objects.

VI. The ACDSense Scenario

To evaluate the feasibility and federation capabilities of the solution described above, we developed ACDSense, an inter-organizational sensor network scenario spanning our three universities: RWTH Aachen University, TU Dresden, and BTU Cottbus-Senftenberg.

Each of our sites contributed heterogeneous sensor implementations, publishing mostly weather-related data. Additionally, each of our sites employed different types of client applications for accessing and controlling sensors, ranging from standard general-purpose XMPP clients over custom mobile apps to Web-based widget dashboards. Each organization runs its own XMPP server in federation with the other institutions' XMPP servers, as shown in Figure 3. Hence, any client can use any server as an entry point to the federated network and access sensors connected to any other server. The following subsections provide detailed descriptions of sensor and client implementations for each site.

A. High-Performance Commodity Sensors and Widget Dashboard in Aachen

The sensor part at RWTH Aachen university represents a class of low-cost high-performance commodity hardware sensors. It is implemented using a Raspberry Pi with a USB-connected thermometer. WLAN and XMPP connections to a sensor MUC room are established automatically at boot time via a configurable Python script. Once connected, the script periodically fetches data from the thermometer and pushes sensor event payloads (cf. Section V-B) into the MUC room. In addition, users can adjust the sensor's update frequency by sending remote command messages to the sensor MUC room.

The client part at RWTH Aachen university represents a class of Web-based clients, accessible from regular modern Web browsers. For ACDSense, we created a sample collaborative Web application, providing a well-defined context for multiple users to monitor and control sensors together. The application is built on top of the ROLE SDK (cf. [13]), providing so called spaces as collaboration contexts. It uses XMPP over WebSocket as a modern transport from Web applications. To support the ACDSense scenario, we built a space populated with three widgets (cf. Figure 3, lower left screenshot). The *Sensor List* widget displays the results of a sensor discovery across the ACDSense federated network and provides buttons for connecting to individual sensors. The *Sensor Meter* widget visualizes live sensor data from all connected sensors in real-time. A *Sensor Map* widget visualizes sensor geo-locations available from sensor metadata. The built-in chat space is also powered by XMPP.

B. Mobilis-based Sensing Apps in Dresden

The sensor part in Dresden consists of a *Sensor Data Generator*. It processes historical hurricane weather data and sends it to multiple sensor MUC rooms. One MUC room represents one weather station equipped with sensors for measuring wind, humidity, ambient temperature, and others.

The client part in Dresden represents a class of native mobile apps for accessing and controlling sensors via XMPP. We developed a native iOS sensing app on top of the Mobilis framework [14] (cf. lower middle of Figure 3). Mobilis enables to write compounds of native mobile apps and cloud services which are dynamically deployed in a runtime environment. The communication is based on XMPP and thus compatible with other XMPP-based approaches. The sensing app allows to do basic sensor discovery, browse sensor domains, select sensors of interest within each domain, and visualize sensor data.

C. Integration of Constrained Devices in Cottbus

The sensor part in Cottbus represents a class of heavily constrained sensor devices. In particular, we show the integration of constrained devices using the Chatty Things approach with the XMPP extension *Temporary Subscription for Presence (TSP)* in our architecture. Zolertia Z1 nodes, running a minimized and modular XMPP stack on top of the Contiki OS, push their sensor data directly to MUC rooms of an XMPP server.

The Chatty Things use TSP to avoid presence overhead when joining a MUC room. To further reduce XMPP message size, we avoid repeated transmission of redundant data, such as sensor metadata. The topic of the sensor MUC room represents the type of the sensor, the first message after joining the MUC room transmits geo-location and the unit of the sensed data. All other messages only transmit the sensed value in a short format. Users can interactively adjust the sensor's update frequency by sending remote command messages to the sensor MUC room.

The client part at BTU Cottbus-Senftenberg demonstrates the feasibility of using standard XMPP client software for accessing sensors. The rightmost lower screenshot in Figure 3 shows the output of a sensor MUC room in an ordinary XMPP client like Adium/Pidgin. The information is human-readable. Thus, no additional software is necessary to monitor Chatty Things sensors, if the JIDs of all sensor MUC rooms are known in advance.

D. Administrative Challenges

The ACDSense scenario was realized within four weeks and then tested and evaluated over several weeks. Before we present detailed evaluation results in Section VII, we share concrete experiences, mostly related to administrative challenges while setting up federation among our three sites.

First, each of our XMPP servers had to be configured for accepting server-to-server connections. As these connections run over port `5269`, and not the standard XMPP client port (`5222`), new firewall filters had to be added at all three sites. In addition, a second DNS SRV record (`_xmpp-server._tcp`) per XMPP server is needed in theory. In practice, most self-hosted XMPP servers use their URL as their XMPP service domain. In this case, an ordinary DNS A or AAAA record is sufficient for clients/servers to connect to the XMPP server, which worked for our setup. Another challenge emerged from the use of the XMPP Multi-User Chat extension. An XMPP server can host multiple MUC services, each of them accessible via an own sub-domain like `conference.xmpp-server.org`. As administrators normally forbid to resolve all sub-domains with a single DNS A record, the MUC sub-domain needs to be added as a DNS alias as well. If it is not possible to modify DNS entries for administrative reasons, a simple workaround is to modify the `etc/hosts` file of each machine taking part in the federation. This worked well for some servers in our small federation scenario described above. Large scenarios certainly require DNS entries for each server and MUC sub-domain. Fixing these administrative issues and sticking to the agreed sensor metadata and data formats (cf. Section V-B) was sufficient to enable inter-organizational sharing of sensor data. Each of our three clients, connected to its home server, was able to discover, access and control sensors running at all three sites.

VII. Evaluation

To further demonstrate the efficiency, in particular the scalability, of our approach, we developed an evaluation infrastructure capable of running different types of experiments. These were mainly targeted at the comparison of individual properties of the MUC-based transport, such as message delivery time, message overheads and message throughput. In the following sections, we describe our evaluation environment and present the results of our experiments.

A. Evaluation Environment

We used three physical machines to run the tests and collect test results. Node 1 and Node 2 each run an OpenFire XMPP server and are coupled using XMPP federation. Node 3 emulates a large number of data senders (i.e., sensors) and heterogenous receivers (i.e., clients).

Sensor data exchanged between the nodes consisted of weather data from [15]. We selected the 1000 most frequent senders out of a total set of 9,503 weather stations. The dataset produced by these stations comprises a time period of 10 days in August 2005 (during Hurricane Katrina).

The Java-based *Data Player* creates and joins a MUC for each weather station and posts sensor data with configurable time acceleration. The *Data Receiver* discovers and joins MUCs to receive sensor data. The *Smack XMPP* library was used for both the senders and Java receivers while we also did tests with Web-based receivers using the JavaScript library *strophe.js*. Both Data Player and Data Receiver were executed on the same machine to avoid time synchronization problems in the collected measurement data. We used a time acceleration factor of 500 to speed up the tests and to stress-test the infrastructure.

For each message on both sender and receiver side, we recorded send/receive times, overall message and sensor data payload sizes as well as sender/receiver JIDs. By default, each message contained a unique message identifier, thus allowing to join data for sent and received messages. We also measured the time to discover large numbers of sensors across the federated environment.

To create reproducible results, we did the tests in this section with a setup of two federated servers in the same LAN. We double-checked part of these results with tests on a true federated setup between Aachen and Dresden. Apart from approximately 20 ms RTT added to message delivery time the results are the same for the LAN setup.

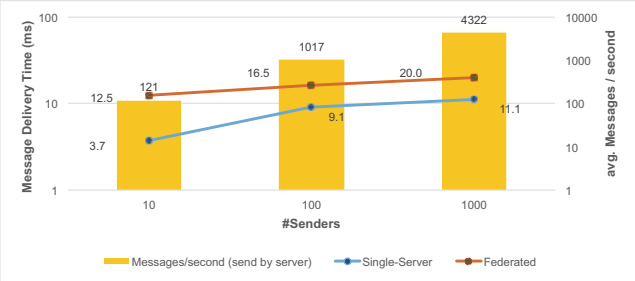

Fig. 4. Message delivery time for varying topologies and numbers of sensors

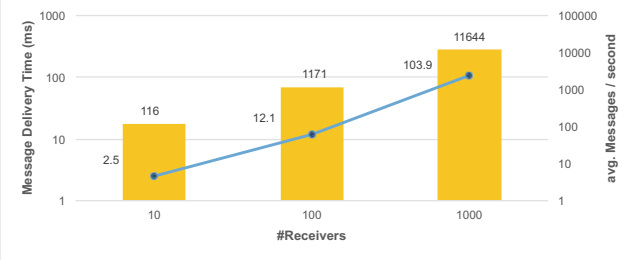

Fig. 5. Message delivery time for varying numbers of receivers per sensor

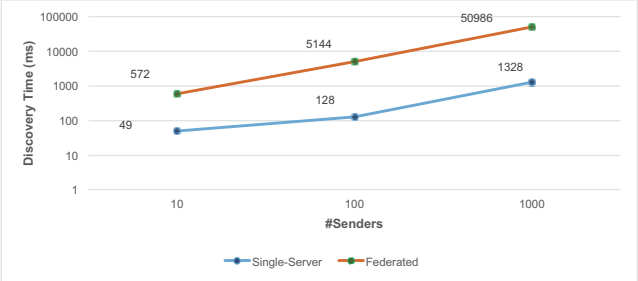

Fig. 6. Sensor discovery time for varying numbers of sensors

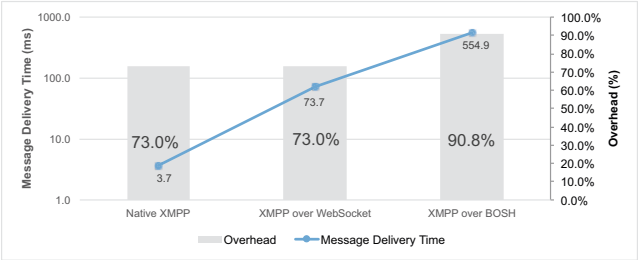

Fig. 7. Message delivery time as well as overhead for native and Web-based XMPP transports

B. Message delivery time

In the first experiment, we connected a single receiver to every MUC that receives the posted sensor data of each emulated weather station. For 10 senders, the XMPP Server had to handle 20 sessions: 10 for the senders and 10 for the receivers.

We compared message delivery time using 10, 100 and 1000 senders, respectively (compare Figure 4). Message latency only slightly increases with the load on the server (i.e., messages per second). Even with more than 4,000 messages per second, the delivery time is 11.1 ms if using one XMPP server. The federation overhead when using two servers is only about 10 ms and does not increase with the load.

In a second experiment, we evaluated the effect of the number of receivers per MUC. We ran a series of tests with the most frequent sender station posting to only one MUC, which is then joined by 10, 100, and 1000 clients, respectively. The total throughput of messages for the 1:1000 case is even higher (11,644 messages per second) than for the 1000:1 case (4,322 messages). This is due to the fact, that only the weather station with the most frequent events was used in the test.

The increase in message delivery time is again proportional to the total number of messages sent per second (cf. Figure 5). Moreover, the numbers at certain throughput levels are comparable to the first experiment. It hence does not matter whether the number of senders or the number of receivers is increased. Message delivery time grows in both cases.

As the average message size was 385 Bytes, we transported an impressive workload of 42 MBit/s upstream and 4.1 MBit/s downstream in the 1:1000 test case. Our Intel i7 Quadcore XMPP server ran at 12-18% CPU load in this test. This proves the scalability of the XMPP MUC approach.

C. Discovery

In experiment 1, we also measured the time to retrieve the descriptions of all weather stations (worst case performance). Unfortunately, the time needed to fetch the descriptions grows linearly with the number of MUCs (cf. Figure 6). This is due to the fact that the description can only be determined if an individual `disco#info` request is issued for each MUC. This results in 1,000 request/response cycles for the 1000:1 case.

While the time for a single XMPP server is still acceptable (1.3 s), the federated discovery is only usable up to 100 rooms, as there is already 5 s to wait for the result. A discovery request with 1000 weather stations in the federated case takes 50 s and thus becomes unusable.

However, if only a list of MUCs and their JIDs is needed, a single `disco#items` query is sufficient. Regardless of the number of senders, this resulted in a discovery time of around 80 ms, both for single server and federated servers.

D. Native vs. Web-based XMPP

In another experiment, we compared the performance of native XMPP (baseline) vs. Web transports to a local server. For the latter, we compared both XMPP over WebSocket and XMPP over BOSH. We ran the test described in experiment 1 and measured the average message delivery times and message overhead. The results are shown in Figure 7: message delivery time of the WebSocket connection increased to 73.7 ms and even 554.9 ms for BOSH, compared to 3.7 ms over the native TCP connection. The overhead for the WebSocket remained at the 73% baseline, while it increased to 90.8% for BOSH. BOSH uses costly long-polling HTTP connections for message delivery, hence explaining its high overhead. In contrast, the overhead of WebSocket is equal to native XMPP connections. Average message delivery times differ in orders of magnitude.

For both Web transports, message delivery time remains relatively high, since the messages take another indirection over the browser's JavaScript API.

Given its low overhead and acceptable message delivery time, XMPP over WebSocket replaces XMPP over BOSH to realize an effective and efficient bridge between XMPP-based sensor networks and related Web applications. The XMPP community is currently active in stabilizing respective standard drafts. As WebSocket support will probably not be realized in legacy browsers, XMPP over BOSH remains as a fallback solution.

E. Experiments with Short Format

In order to assess the performance of our approach in constrained environments, especially in 6LoWPANs, we created a second experimental setup. Our performance test reconstructed a test case (e.g., temperature reception) of RESTful Web services [16, Sec.6.3]: a one-hop network (IPv6) with Zolertia Z1 wireless sensor nodes. One node runs the 6LoWPAN border router and is connected via the Serial Line Internet Protocol (SLIP) to a computer.

The Chatty Thing pushes its sensed temperature data to the MUC room with the sensor-specific topic `'temp'`. For CoAP, the provided REST example (e.g., `get 'hello world'`) of Contiki OS was used. The completion time (summarized in Table II) for the REST-based approaches was measured as the time between requesting a service and getting its sensed data.

The measurements of XMPP also include the performance for presence status and one-to-one chat message exchanges, to give a general performance overview of typically used XML message stanzas. The very low message size of CoAP is a clear advantage in 6LoWPANs, but XMPP can achieve a comparable performance when the XML message fits in a single TCP/IP packet (i.e., max. 48 Bytes). XMPP can thus be used more efficiently if XML data compression implementations like EXI are available for constrained devices.

TABLE II. REST VS XMPP IN 6LOWPANS

Approach	Request Size	Response Size	Completion Time
RESTful Web Services	85 Bytes	141 Bytes	440 ms
CoAP	15 Bytes	19 Bytes	54 ms
XMPP (Group Chat)	-	144 Bytes	200 ms
XMPP (One-to-One Chat)	-	96 Bytes	122 ms
XMPP (Presence)	-	35 Bytes	38 ms

VIII. CONCLUSIONS

We demonstrated a concept, use case and evaluation of federated data sharing in the Internet of Things using XMPP. Our MUC-based concept complements other approaches and is especially easy to implement and deploy.

As the evaluation results show, XMPP MUC as a transport layer is able to deliver messages within milliseconds in single server and federated settings. With the Chatty Things extensions, it also offers competitive performance on resource constrained devices. Web-based applications are also able to receive the data in reasonable time (especially with XMPP over WebSockets). Other built-in features like presence and access control offer additional value for real-world deployments.

The main drawback still lies in the performance of the discovery, which does not scale well, especially in the federated case. Additional mechanisms are needed, which are currently elaborated in the XMPP community (refer to [17]) with the help of one of the authors of this paper.

REFERENCES

[1] S. Jensen, "Mobile Apps Must Die," http://jenson.org/mobile-apps-must-die, 2011.

[2] P. Saint-Andre, "XMPP: Lessons Learned from Ten Years of XML Messaging," *IEEE Communications Magazine*, vol. 47, no. 4, pp. 92–96, 2009.

[3] Z. Jaroucheh, X. Liu, and S. Smith, "An Approach to Domain-based Scalable Context Management Architecture in Pervasive Environments," *Springer Personal and Ubiquitous Computing Journal*, vol. 16, no. 6, pp. 741–755, 2012.

[4] M. Kuna, H. Kolaric, I. Bojic, M. Kusek, and G. Jezic, "Android/OSGi-based Machine-to-Machine Context-aware System," in *11th IEEE Conference on Telecommunications (ConTEL)*, 2011.

[5] A. Rowe, M. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. Garrett, J. Moura, and L. Soibelman, "Sensor Andrew: Large-scale Campus-wide Sensing and Actuation," *IBM Journal of Research and Development*, vol. 55, no. 1.2, pp. 6:1–6:14, 2011.

[6] XMPP Standards Foundation, "Tech pages/IoT XepsExplained," http://wiki.xmpp.org/web/Tech%20pages/IoT%20XepsExplained, 2014.

[7] A. Hornsby and E. Bail, "μXMPP: Lightweight Implementation for Low Power Operating System Contiki," in *IEEE Conference on Ultra Modern Telecommunications & Workshops (ICUMT)*, 2009.

[8] R. Klauck and M. Kirsche, "Chatty Things - Making the Internet of Things Readily Usable for the Masses with XMPP," in *8th IEEE Conference on Collaborative Computing: Networking, Applications & Worksharing (CollaborateCom)*, 2012.

[9] Z. Shelby, K. Hartke, and C. Bormann, "Constrained Application Protocol (CoAP)," IETF, Internet-Draft, 2013. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-core-coap/

[10] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An application protocol for billions of tiny internet nodes," *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, 2012.

[11] Eurotech and IBM, "MQTT v3.1 Protocol Specification," http://www.ibm.com/developerworks/webservices/library/ws-mqtt/, 2010.

[12] A. Stanford-Clark and H. L. Truong, "MQTT for sensor networks (MQTT-SN) protocol specification version 1.2," http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf, 2013.

[13] D. Kovachev, D. Renzel, P. Nicolaescu, and R. Klamma, "DireWolf - Distributing and Migrating User Interfaces for Widget-Based Web Applications," in *13th Int. Conference on Web Engineering (ICWE)*, 2013.

[14] D. Schuster, R. Lübke, T. Springer, and A. Schill, "Mobilis - Comprehensive Developer Support for Building Pervasive Social Computing Applications," in *Networked Systems (NetSys)*, 2013.

[15] Kno.e.sis Project Consortium, "Linked Sensor Data," http://wiki.knoesis.org/index.php/LinkedSensorData, 2010.

[16] D. Yazar and A. Dunkels, "Efficient Application Integration in IP-based Sensor Networks," in *Proc. of the 1st ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)*, 2009, pp. 43–48.

[17] P. Waher and R. Klauck, "XEP-0347: Internet of Things - Discovery," http://xmpp.org/extensions/xep-0347.html, April 2014.